Integer Factorization Based on Elliptic Curve Method: Towards Better Exploitation of Reconfigurable Hardware

Giacomo de Meulenaer, François Gosset, Guerric Meurice de Dormale[*] Jean-Jacques Quisquater

UCL/DICE Crypto Group, Place du Levant, 3, B-1348 Louvain-La-Neuve, Belgium
E-mail: {demeule,gosset,gmeurice,quisquater}@dice.ucl.ac.be

Abstract

Currently, the best known algorithm for factorizing modulus of the RSA public key cryptosystem is the Number Field Sieve. One of its important phases usually combines a sieving technique and a method for checking smoothness of mid-size numbers. For this factorization, the Elliptic Curve Method (ECM) is an attractive solution. As ECM is highly regular and many parallel computations are required, hardware-based platforms were shown to be more cost-effective than software solutions.

The few papers dealing with implementation of ECM on FPGA are all based on bit-serial architectures. They use only general-purpose logic and low-cost FPGAs which appear as the best performance/cost solution. This work explores another approach, based on the exploitation of embedded multipliers available in modern FPGAs and the use of high-performances FPGAs.

The proposed architecture – based on a fully parallel and pipelined modular multiplier circuit – exhibits a 15-fold improvement over throughput/hardware cost ratio of previously published results.

Keywords: Factorization, elliptic curve, FPGA, parallel modular multiplier.

1 Introduction

With the advent of public key cryptography, many useful functionalities appeared such as digital signature, public key encryption, key agreement, ... For those needs, the most deployed scheme remains RSA, co-invented in 1977 by R. Rivest, A. Shamir and L. Adleman [16]. The security of this cryptosystem relies on the intractability of the factorization of big composite integers. This mathematical hard problem experiences therefore a renewed interest. It is believed that 1024-bit RSA keys are sufficiently big to sustain today's attacks.

Currently, the best known algorithm for factorizing RSA modulus is the Number Field Sieve (NFS), introduced by Pollard in 1991. It is composed of a sieving and a matrix step, the former being the most expensive part for 1024-bit keys [6]. The reader is referred to [15] for an introduction to the NFS and to [10] for the details. This paper focuses on the sieving step and more precisely on the relation collection step. This task is usually performed by a combination of a sieving technique and a method for factorizing mid-size numbers [8]. While those numbers are easily factorizable, the challenge lies in the amount of computation: factorization of 10^{14} 125-bit numbers for a 1024-bit modulus is required (using [3]). For this task, the Elliptic Curve Method (ECM) appears as an attractive solution.

Up to now, the best successful factorization attempts were for RSA-200 (663-bit) and RSA-640, solved in 2005 by Bahr, Boehm, Franke and Kleinjung. Beside those software-based solutions, other propositions related to special purpose hardware came out to lower the cost of both machine and power consumption. Such proposals in the context of the NFS sieving step are the TWINKEL device, mesh-based sieving, TWIRL and SHARCS (see [17] for an overview and references). ECM is one of the building block of SHARCS [3] and it was proposed to combine it with TWIRL in [5].

For the design of a hardware machine a platform has first to be chosen. The two main possibilities are ASICs (application-specific integrated circuits) or FPGAs (field programmable gate arrays). While ASICs have potentially the highest performance per transistor and the best power consumption, the achievement of a real and working device is a long and very expensive process. FPGAs have the advantage of adding an extra layer of abstraction during the design. This results in low non-recurring engineering (NRE) costs and a low development-production time. Together with their increasing area & speed and falling prices & power consumption, FPGAs are highly prized in the context of cryptanalysis. Even if it appears that taking *ALL* the costs into account, ASICs are less expensive than FPGAs, they are at least an interesting development platform.

[*] Supported by the Belgian fund for industrial and agricultural research

This paper is structured as follows: Section 2 presents previous works on hardware implementation of ECM and explains our contribution. Then, Section 3 reminds the mathematical background of elliptic curves and ECM. The choices about the arithmetic circuits necessary for implementing ECM are discussed in Section 4. The proposed ECM architecture stands in Section 5 and implementation results together with a quick cost assessment for factorizing 1024-bit modulus are provided in Section 6. Finally, the conclusion and a few words about further works are given in Section 7.

2 Previous Works and Contribution

The motivation of this work is to assess the cost of an ECM processor as a support for the NFS algorithm. As sketched in the introduction, this work focusses on a FPGA-based ECM implementation. The aim is to show that embedded multipliers of modern FPGAs are a key advantage for ECM and that current high-performance FPGAs outperforms the throughpupt/cost ratio of low-cost FPGAs.

If the required number of ECM processors makes the use of ASICs cheaper, the proposed FPGA design can still be migrated to ASIC using standard library and IPs for multipliers and RAMs.

2.1 Hardware Implementations

The first published hardware implementation of ECM was proposed by Pelzl, Šimka et al. in 2005 [14]. The aim of this circuit was to check the smoothness of sieving reports of the SHARK device [3]. It is formed by a collection of parallel ECM units in a Xilinx FPGA together with an external ARM microcontroller. Each unit embeds a memory, a controller and an ALU able to perform addition/subtraction and radix-2 Montgomery multiplication [11]. Carry propagate adders were chosen. An improved pipelined version of this design was later used in [5].

This proof of concept was deeply improved by Gaj et al. in 2006 [4]. They removed the external microcontroller and improved the operating frequency and the Montgomery multiplier. They use two multipliers and one adder/subtracter in parallel in each ECM unit. Carry save adders were chosen. Their conclusion is that, compared with high-performances FPGAs and general purpose processors, low-cost FPGAs are the best choice for implementing ECM.

2.2 Contribution

The common feature of all the previous works is the use of a bit-serial by parallel architecture. All the arithmetic circuits are built with the general purpose logic of FPGAs.

Modern FPGAs have interesting features including embedded multipliers. It makes therefore sense to try to exploit them. For instance, Virtex4SX devices embed 128, 192 or 512 DSP48 blocks able to perform 17×17 unsigned multiply and accumulate/shift-and-add. Their low-cost counterparts, the Spartan3 devices, are equipped with up to 104 17×17 unsigned multipliers. Hardware cost plays an important role here and fortunately, high-performances devices like Virtex4 are no longer much more expensive than low-cost FPGAs such as Spartan3. Together with their increased speed, enhanced functionalities of DSP blocks and their number of multipliers/$, Virtex4SX are therefore potentially the sweet spot for the ECM application.

Our contribution is to show that building the core component – the modular multiplier – with fully pipelined embedded multipliers of high-performance FPGAs leads to a highly efficient ECM processor. It is an order of magnitude better than the low-cost solution presented in [4] with respect to the cost/performance ratio. To exhibit the full power of this approach, a parallel architecture with a throughput of 1 multiplication per clock cycle was chosen.

The main motivation of this work is to propose a much more efficient ECM engine for the SHARK device [3]. Factorization of 125-bit numbers is therefore required and a throughput of 10^{14} factorizations over a year is needed for the 1024-bit modulus.

3 Elliptic Curve Method

The Elliptic Curve Method is a probabilistic method for integer factorization which uses elliptic curves. It is the best known method to factorize mid-sized numbers together with the Multiple Polynomial Quadratic Sieve (MPQS) [2]. ECM seems to be the best choice for hardware implementation since it is highly regular, not too I/O intensive and many parallel computations are required [14].

3.1 Elliptic Curves

Let $\mathbb{Z}_p$ be the set of integers $\{0, 1, \ldots, p-1\}$ with a prime $p > 3$ with addition and multiplication (mod p). A non-singular elliptic curve (EC) E over $\mathbb{Z}_p$ is defined as a set of points $(X, Y) \in \mathbb{Z}_p \times \mathbb{Z}_p$ satisfying the reduced Weierstrass equation, together with the point at infinity O:

$$Y^2 = X^3 + aX + b$$

where $a, b \in \mathbb{Z}_p$ and $4a^3 + 27b^2 \neq 0$. The elliptic curve E together with an addition law admits a group structure where the point at infinity O is the neutral element.

The addition of two points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ (assuming that $P, Q \neq O$) gives $R = (x_R, y_R)$

in most cases through the following computations:

$$\left\{ \begin{array}{lcl} x_R & = & \lambda^2 - x_P - x_Q \\ y_R & = & \lambda(x_P - x_R) - y_P \end{array} \right.$$

$$\text{with } \lambda = \left\{ \begin{array}{ll} (y_Q - y_P)/(x_Q - x_P) & \text{when } P \neq \pm Q \\ (3x_P^2 + a)/2y_P & \text{when } P = Q \end{array} \right.$$

The definition of O solves the problem in the computation of λ when P is added to its inverse $-P = (x_P, -y_P \mod p)$. The result is defined as $P + (-P) = O$.

To speed up the computations, it is more efficient to use elliptic curves with Montgomery form (cf. Section 3.4) :

$$bY^2 = X^3 + aX^2 + X \qquad (1)$$

where $a, b \in \mathbb{Z}_p, a^2 \neq 4$ and $b \neq 0$. When using homogeneous (projective) coordinates instead of affine coordinates, Equation 1 becomes:

$$by^2z = x^3 + ax^2z + xz^2. \qquad (2)$$

The triple $(x : y : z)$ represents $(\frac{x}{z}, \frac{y}{z})$ in affine coordinates and $(0 : 1 : 0)$ is the point at infinity O.

In this work, elliptic curves with Montgomery form in homogeneous coordinates are used.

3.2 ECM Algorithm

The elliptic curve method for integer factorization was invented by Lenstra [9]. It is an improvement of the $p - 1$ method of Pollard. Its description follows [12].

The $p - 1$ method tries to find a prime factor p of n. The first phase of the algorithm selects an integer a and computes $b \equiv a^k \mod n$ with $k > 0$ divisible by all prime powers below a bound B_1. If $p - 1$ divides k, then p divides $b - 1$ by Fermat's little theorem ($a^{p-1} \equiv 1 \mod p$ if $\gcd(a, p) = 1$). Let $d = \gcd(b - 1, n)$. If $1 < d < n$, then d is a factor of n. If $d = n$, k must be reduced. The first phase fails if $d = 1$. In this case, the computation can be continued with the second phase. For this purpose, a second bound $B_2 \gg B_1$ is selected and it is assumed that $p - 1 = qs$, where q divides k and s is a prime between B_1 and B_2. If $t \equiv b^s - 1 \equiv a^{ks} - 1 \mod n$ then p divides t by Fermat's little theorem, since $p - 1$ divides ks. The problem is to find s, and hence p.

The $p - 1$ method can be seen as an attempt to find the neutral element (i.e., 1) of the multiplicative group of the nonzero elements of $\mathbb{Z}_p$. It fails if $p - 1$, i.e. the number of elements (group order), cannot be written as a product of prime powers smaller than B_1 and at most one prime between B_1 and B_2. ECM avoids this by considering the group of an elliptic curve over $\mathbb{Z}_p$, where the number of elements randomly varies. As a result, if the method fails with one curve, another curve can be chosen with hopefully a group order satisfying the required property.

In the ECM algorithm (Algorithm 1), the computations are done modulo n as if $\mathbb{Z}_n$ was a field. If the point computed at the end of the phase 1 is the point at infinity O of E over $\mathbb{Z}_p$, then its coordinate z equals $0 \mod p$ and $\gcd(z, n)$ can lead to p. The operation kP is defined as $P + \cdots + P$ (k times) and called scalar multiplication.

If $Q = kP \neq O$, k might miss just one large prime factor to divide the group order (as in Pollard $p - 1$ method). Phase 2, also called continuation, tries every prime p in the range $[B_1, B_2]$. There are several ways to perform this task [12]. The standard continuation was chosen here and seems to be the most suitable continuation for a hardware implementation. Its description follows [4].

Algorithm 1 ECM factorization

```
Input: $n$ to be factored, arbitrary curve $E$, point $P \in E$ over $\mathbb{Z}_n$,
bounds $B_1 < B_2 \in \mathbb{N}$
Output: $d$: factor of $n$, $1 < d \leq n$, or Fail
Phase 1:
  $k \leftarrow \prod_{p \in \mathbb{P}, p \leqslant B_1} p^{e_p},\ e_p \leftarrow \max\{q \in \mathbb{N} : p^q \leqslant B_2\}$
  $Q = (x_q, y_q, z_q) \leftarrow kP,\ d \leftarrow \gcd(z_q, n)$
  if $d > 1$ then    return $d$
  else               goto phase 2.
Phase 2:
  $t \leftarrow 1$
  for each $p \in \mathbb{P}$ such that $B_1 < p < B_2$ do
    $(x_{pQ}, y_{pQ}, z_{pQ}) \leftarrow pQ$
    $t \leftarrow t \cdot z_{pQ} \mod n,\ d \leftarrow \gcd(t, n)$
  if $d > 1$ then    return $d$
  else               Fail (Try with another curve).
```

The standard continuation (Algorithm 2) avoids computing every pQ by using a parameter $0 < D < B_2$ in order to express each prime $B_1 < p < B_2$ in the form $p = mD \pm j$ with $j \in \left[0, \lfloor \frac{D}{2} \rfloor\right]$ and m varying to cover all the primes in the interval $[B_1, B_2]$. Since p is prime, $\gcd(j, D) = 1$.

To speed up the computations, a usual way to proceed is to precompute a table T of the multiples jQ of Q. Then, the sequence of the points mDQ is computed and the product of every $x_{mDQ} \cdot z_{jQ} - x_{jQ} \cdot z_{mDQ}$ for which $mD \pm j$ is prime is accumulated. It is indeed possible to show (see [14]) that $pQ = O$ if and only if $x_{mDQ} \cdot z_{jQ} - x_{jQ} \cdot z_{mDQ} \equiv 0 \mod d$ (d is an unknown prime factor of n).

3.3 Parametrization

The different values and parameters of this work were chosen as in the work of Šimka et al. [14] and Gaj et al. [4]. More precisely, same values for B_1 and B_2 are used: $B_1 = 960$ and $B_2 = 57000$. They were chosen to find factors of up to around 40 bits. The parametrization of Suyama [20] is also employed for the computation of initial points and curve coefficients (a, b of Equation 2).

Algorithm 2 Standard Continuation (Phase 2)

```
Input: $n$, $E$, $Q$ (result of phase 1), $B_1 < B_2 \in \mathbb{N}$ and $D$
Output: $d$: factor of $n$, $1 < d \leq n$, or Fail
Precomputations:
    $M_{min} \leftarrow \lfloor (B_1 + \frac{D}{2}) / D \rfloor, \quad M_{max} \leftarrow \lceil (B_2 - \frac{D}{2}) / D \rceil$
    Table $J$:    $j \in \{1, \ldots, \frac{D}{2}\}$ s.t. $\gcd(j, D) = 1$
    Table $MJ$:   $MJ[m, j] \leftarrow 1$ if $mD \pm j$ is prime, else $0$,
                  with $j \in J$ and $M_{in} \leq m \leq M_{max}$
    Table $T$:    $\forall j \in J$, coordinates $(x_{jQ} :: z_{jQ})$ of points $jQ$
Main Computations:
    $t \leftarrow 1, \quad Q_D \leftarrow DQ, \quad Q_m \leftarrow M_{min}Q$
    for $m = M_{min}$ to $M_{max}$ do
      for $j \in J$ do
        if $MJ[m, j] = 1$ then
          retrieve $jQ$ from table $T$
          $t \leftarrow t \cdot (x_{Q_m} \cdot z_{jQ} - x_{jQ} \cdot z_{Q_m})$
      $Q_m \leftarrow Q_m + Q_D$
    return $d \leftarrow \gcd(t, n)$
```

Ideally, D should be taken near $\sqrt{B2}$ in order to minimize the computations for the table T and the sequence of points Q_m. D must also be small enough and with many prime factors in order to reduce the number of all possible j (since $\gcd(j, D) = 1$) hence the size of the table T. We chose $D = 2 \cdot 3 \cdot 5 \cdot 7 = 210$ which is close to $\sqrt{B2} = 239$, leading to 24 possible values for j.

3.4 Curve Arithmetic

With elliptic curve in Mongtomery form and homogeneous coordinates, the addition and doubling of points can be done without the costly modular division. The intermediate computations of these operations do not use the y coordinate (cf. Algorithm 3), which can be recovered from the other coordinates. Nevertheless, this is not necessary as ECM does not use y coordinates.

Algorithm 3 Point add and double for Montgomery curves

```
$x_{P+Q} \leftarrow z_{P-Q}[(x_P - z_P)(x_Q + z_Q) + (x_P + z_P)(x_Q - z_Q)]^2$
$z_{P+Q} \leftarrow x_{P-Q}[(x_P - z_P)(x_Q + z_Q) - (x_P + z_P)(x_Q - z_Q)]^2$
$4x_P z_P \leftarrow (x_P + z_P)^2 - (x_P - z_P)^2$
$x_{2P} \leftarrow (x_P + z_P)^2 (x_P - z_P)^2$
$z_{2P} \leftarrow (4x_P z_P)[(x_P - z_P)^2 + a_{24}(4x_P z_P)]$   (with $a_{24} = \frac{a+2}{4}$)
```

In Algorithm 3, the addition $P + Q$ uses the coordinates of the point difference $P - Q$.

The scalar multiplication can be efficiently computed with the Montgomery ladder Algorithm (4). The additions in this algorithm can be computed with Algorithm 3 since the point difference always equals P_0. If $z_{P_0} = 1$ (e.g. after normalization), the number of multiplications of each step of the algorithm decreases from 11 to 10 (squarings are performed with multiplications).

Algorithm 4 Montgomery ladder

```
Input: Point $P_0 \in E, \quad k = (k_{s-1}k_{s-2} \ldots k_1 k_0)_2 > 0$
Output: $Q = kP_0$.
  $Q \leftarrow P_0, \quad P \leftarrow 2P_0$
  for $i = s - 2$ downto $0$ do
      if $k_i = 1$ then    $Q \leftarrow P + Q, \quad P \leftarrow 2P$
      else                 $P \leftarrow P + Q, \quad Q \leftarrow 2Q$
  return $Q$
```

In phase 2, the sequence $Q_m = Q_m + Q_D$ can also be computed with Algorithm 3 because the difference $Q_m - Q_D$ equals the previous result Q_{m-1} (for the first addition, $Q_{M_{min}-1}$ is precomputed).

4 Arithmetic Circuits

The effectiveness of the ECM processor lies in an efficient area-time (AT) arithmetic unit. In particular, the modular multiplication plays a crucial role.

This work implies the use of a Xilinx's Virtex4SX FPGA. To ensure a high operating frequency and scalability, each combinatorial operation is pipelined as much as necessary. A digit size of 17 bits is chosen since it fits with the 17×17 unsigned multipliers and the dedicated shift-and-add functionality of the DSP48. Pipelined carry propagate adders are also used to take advantage of the available fast carry chain. It is also consistent with multipliers' input. The design is therefore both pipelined *horizontally* (e.g. the carry chain) and *vertically* (e.g. between two adders).

4.1 Parallel versus Serial

Now that embedded multipliers are used, it is still necessary to choose the type of architecture. As the modular multiplication algorithms are typically iterative and regular, the two main choices are an iterative (looped) or a parallel (fully unrolled) architecture.

A good choice for an iterative architecture is a digit-serial by parallel multiplier. It can be built with a pipelined row of $\lceil \log_{2^{17}} n \rceil$ multipliers (e.g. [18]). For this work, it means a 17×136 multiplier is required. The computation loops over this multiplier and alternately processes a multiplication by a digit and the modular reduction. The advantage of this methodology is that resources grow linearly with the operand width. The disadvantage is its *worst-case* behavior: area (or time, caused by extra iterations) has to be spent to compute any irregular operation.

Another choice is a parallel multiplier. Both operands enter in parallel and one multiplication is computed each clock cycle. The high latency causes no data dependency problems as there are many factorizations to perform at the same time. Such circuit can be built with roughly

$2\lceil \log_{2^{17}}(n) \rceil^2$ multipliers. For this work, it means 16 cascaded 17×136 multipliers, or in other words 2 interleaved 136×136 multipliers, are required. Economy of scale can be achieved by the use of sub-quadratic algorithms for the multiplication[1]. However, our preliminary implementation results show that the improvement is not worth the effort for considered bit-sizes. A parallel circuit has several advantages: the circuit can be data-driven – removing the need for control logic – and fully specialized for each operation of the algorithm. Moreover, data loading and unloading are simpler compared to a set of iterative architectures. As a result this solution exhibits the best AT product. The disadvantage is the flexibility: the maximum bit-size depends on the number of multipliers in the FPGA. The unused remaining multipliers cannot be directly exploited as well.

To give the best advantage of the multipliers, a parallel architecture was chosen for this work. As shown below, it has many advantages from an AT point of view. With the complexity above, the maximum bit-sizes are 135-bit (SX25), 169-bit (SX35, with 8 LUT-based multipliers) and 271-bit (SX55). In the context of SHARK, a Virtex4 SX25 with 128 DSP multipliers is sufficient. Indeed, a number of digits $d = 8$ provides a 135-bit modular multiplier and requires $2d^2 + 1 = 129$ multipliers. The last multiplier has to be implemented with LUTs. If flexibility is an issue, it is still possible to move to 17×136 iterative modular multipliers. The improvement will still be substantial with respect to a pure LUT-based architecture.

4.2 Modular Multiplier

Montgomery multiplication [11] is an efficient way to perform the modular multiplication, the most important operation in ECM. Indeed, divisions by non power of two are avoided and it mainly involves computations depending on the LSBs. This is consistent with the carry propagation direction of adders and enables pipelining of the carry chain.

The classical Montgomery multiplication algorithm (cf. 5) works mod n and needs a conditional final subtraction as the result is bounded by $2n$. This comparison causes not only extra computations, it forces the complete propagation of the carry pipeline. To avoid this problem, a convenient solution is to work mod $2n$. Provided that $4n < R$, the *without final subtraction* version of the algorithm [19] ensures a bounded output ($< 2n$) if bounded inputs $x, y < 2n$ are applied. This technique is used in this paper.

The Montgomery multiplication works in the Montgomery domain: it computes $xyR^{-1} \mod n$ instead of $xy \mod n$. If the inputs x, y are in the Montgomery domain, $\tilde{x} = xR \mod n, \tilde{y} = yR \mod n$, the output is also in the Montgomery domain: $\tilde{x}\tilde{y}R^{-1} \mod n = xyR \mod n$. All the phases of ECM can therefore be computed in the Montgomery domain, leaving the removal of the R factor at the very end of the computation.

Algorithm 5 Montgomery modular multiplication

```
Input: $n,\ x, y$ with $0 \le x, y < n, R = b^d$ with $\gcd(n, b) = 1$ and $n' = -n_0^{-1} \mod b$.
Output: $xyR^{-1} \mod n$.
    $A \leftarrow 0$
    for $i = 0$ to $(d-1)$ do
        $u_i \leftarrow (a_0 + x_i \cdot y_0) \cdot n' \mod b$
        $A \leftarrow (A + x_i \cdot y + u_i \cdot n)/b$          $(A < 2n)$
    if $A \ge n$ then
        $A \leftarrow A - n$                                  $(A < n)$
    return($A$)
```

In addition to the removal of the final subtraction, a modified version of the Montgomery multiplication introduced by Orup in [13] and called "Montgomery Tail Tailoring" in [7] is used (cf. Algorithm 6). It supposes a radix b (equal to 2^{17}) and inputs represented by their digits: i.e. $n = (n_{d-1} \cdots n_1 n_0)_b$. Compared with the original algorithm of Orup, the last iteration is computed as the classical algorithm to avoid the increase of the dynamic by one digit.

Algorithm 6 Montgomery Tail Tailoring multiplication

```
Input: $n,\ x, y$ with $0 \le x, y < 2n, R = 2b^d$ with $\gcd(n, b) = 1$ and $n' = -n_0^{-1} \mod b$.
Output: $xyR^{-1} \mod 2n$.
Precomputation: $ns = (ns_d \cdots ns_1 ns_0)_b = n \cdot n'$.
    $A \leftarrow 0$
    for $i = 0$ to $(d-2)$ do
        $u_i \leftarrow (a_0 + x_i \cdot y_0) \mod b$
        $A \leftarrow (A + x_i \cdot y + u_i \cdot ns)/b$            $(A < bn + n)$
    $u_{d-1} \leftarrow (a_0 + x_{d-1} \cdot y_0) \cdot n' \mod b$
    $A \leftarrow (A + x_{d-1} \cdot y + u_{d-1} \cdot n)/b$         $(A < 3n)$
    if $A \mod 2 = 1$ then
        $A \leftarrow A + n$                                    $(A < 4n)$
    return($A/2$)                                           $(A < 2n)$
```

Algorithm 6 performs the $d-1$ first iterations with a scaled module $ns = n \cdot n'$ such that its constant $ns' = -ns_0^{-1} \mod b$ is equal to 1. This is always the case since $n' = -n_0^{-1} \mod b$. This simplification saves the $d-1$ digit multiplications by n' previously needed to update u_i in Algorithm 5. ns has one extra digit compared to n but this does not increase the number of digit multiplications since $ns_0 = -1 \mod b = b - 1$ and the least significant digit of $(A + x_i \cdot y + u_i \cdot ns)$ equals 0 (by construction). The update of A in the for loop

$$A \leftarrow (A + x_i \cdot y + u_i \cdot ns)/b$$

[1] This approach supposes to not interleave the multiplication and the modular reduction. Here, this technique is only useful for the multiplication phase as truncated multiplications are used for the modular reduction (cf. Section 4.2).

is therefore computed in practice by:

$$A \leftarrow (A + x_i \cdot y)/b + u_i \cdot (ns_{d..1} + 1),$$

where $ns_{d..1} = ns/b$, meaning that only the digits 1 to d of ns are kept. This input $\widetilde{ns} = ns_{d..1} + 1$ can be precomputed. The total number of digit multiplications is $(d-1) \cdot 2d + 2d + 1 = 2d^2 + 1$ instead of $2d^2 + d$ using the classical Montgomery algorithm.

Our analysis on the bound of the Tail Tailoring Algorithm[2] showed that an extra correction step ($A = A/2 \mod n$) is required to ensure the $x, y < 2n$ precondition.

The circuit of the modular multiplication is presented in Figure 1. The d (8) iterations are implemented with d similar circuits. However, they are slightly different for the first and last iterations: the first is simpler since $A = 0$ and the last one uses n' and n in place of $\widetilde{ns}$. In Figure 1, only one of the $d-2$ identical circuits for the other iterations is represented. Each multiplier executes a $1 \times d$-digit product (17×136). The last multiplier ($\times n'$) can be implemented with general purpose logic, especially as it is a truncated multiplication: only the LSBs are needed in the step $u_{d-1} \leftarrow (a_0 + x_{d-1} \cdot y_0) \cdot n' \mod b$ of Algorithm 6.

The $1 \times d$-digit products (17×136) are implemented with cascaded DSP48s. To deal with the horizontal pipeline behavior of adders, the digits of the operands are temporally shifted, i.e. the next digit is sent one clock cycle after the current one (Least Significant Digit first). This pipelining method does not require extra resources except when the shift size is different from the digit. The building block of the $1 \times d$-digit products (17×136) is sketched in Figure 2 and uses only DPS48's resources.

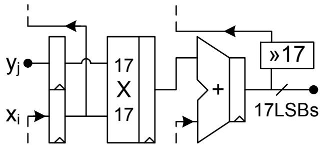

Figure 2. Building block of a $1 \times d$ digits product

Because of the parallel behavior of the multiplier, shift registers are needed to retime and feed the operands throughout the circuit. According to Figure 1, the parallel operand y must be provided in the d circuits implementing the iterations. The same stands for $\widetilde{ns}$ except for the last iteration. For operand x, each iteration consumes a digit. As a result, the width of the shift register decreases with the iteration number (not represented in Figure 1). Inputs n and n' can be inserted after a delay corresponding to the latency of the circuits of the first $d-1$ iterations. For the correction step $A = A/2 \mod n$, it is implemented by a shifter and a conditional adder depending on the LSB of A.

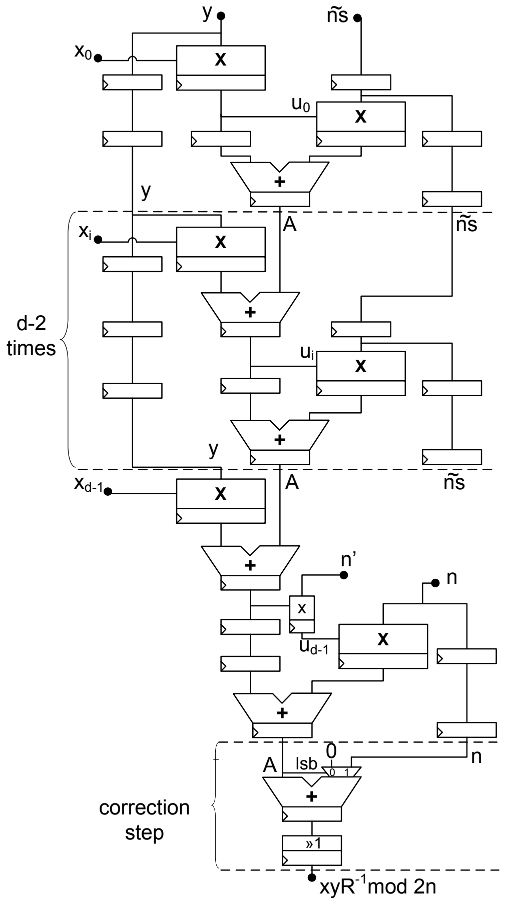

Figure 1. Modular multiplier architecture

Using the parallel approach, circuits for the last iteration and for the correction step maximize the AT product as they are used each cycle. With the serial approach, those irregular operations would have been tackled either with a worst case circuit (like the classical algorithm, loosing the saving of $d-1$ digit multiplications) or with more time (using extra cycles to lift the extra factors).

4.3 Modular Adder/Subtracter

Using a parallel architecture, a modular adder/subtracter is not straightforward. Usually, several comparisons are required to ensure the result of an add/sub $\in [0, n[$. Performing those comparisons would require breaking the temporal shifting behavior of the inputs. A solution to this problem is to make the result positive and divide it by $4 \mod n$.

The add/sub circuit is presented in Figure 3. It uses two's complement representation and assumes $A, B \in [0, 2n[$. The worst case for the addition $(A + B)/4 \mod n$ is

[2] For the bound, it is important to remember the condition $x < 2n$ meaning that $x_{d-1} < b/2 - 1$.

$(A + B + 3n)/4$. The result is therefore bounded by $2n$ as $(4n+3n)/4 < 2n$. For the subtraction the result is bounded as $-2n < (A - B) < 2n$. If $2n$ is added to this result, it also $\in [0, 4n[$. The output is therefore also bounded by $2n$.

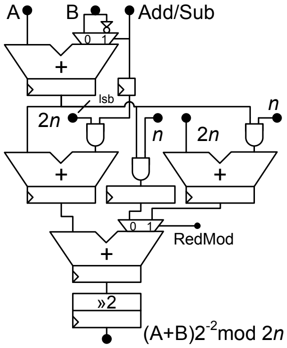

Figure 3. Adder/Subtractor $\mod 2n$

The RedMod control bit is computed as follows:

if $n_0 = 1$ **then** $\text{LSB}_1(A \pm B)$ xor Add/Sub
else $\text{LSB}_1(A \pm B)$ xor Add/Sub xor $\text{LSB}_0(A \pm B)$.

This circuit adds of course an extra 2^{-2} factor. To avoid this problem, all the operands are pre-multiplied by a factor $2^4 \mod n$. This works well with the Montgomery multiplication since $(A2^4R \pm B2^4R)/4 \times (A'2^4R \pm B'2^4R)/4 = C2^2R \times C'2^2R = C2^2R \cdot C'2^2R \cdot R^{-1} = C \cdot C'2^4R$.

5 Proposed ECM Architecture

Based on the arithmetic core developed above, this section presents the whole ECM architecture.

5.1 Architecture Overview

The global ECM architecture is illustrated in Figure 4. It is made of a server (e.g. a PC) and an arbitrary number of clients (for instance FPGAs). The hardware clients take care of the computationally-intensive part of the work: the scalar multiplication. The software server handles the low-throughput operations like precomputations.

Here, the hardware clients are not supposed to perform both phase 1 and 2. As many FPGAs are expected for a 1024-bit factorization, it is not a problem to specialize an FPGA for a given task. Moreover, as a phase 2 is not always necessary, this approach is more convenient from a computational load balancing point of view. The drawback is the increased bandwidth requirements.

5.2 Software Server

Currently, the server computes for each curve the parameter a_{24}, points P_0 and $2P_0$ (for Algorithm 4) and Montgomery constants ns, $\widetilde{ns}$ and n'. P_0 is normalized (dividing x_{P_0} by z_{P_0}) as it is also the point $P - Q$ of Algorithm 3. Nevertheless, as coordinates of points must be pre-multiplied by the Montgomery constant, z_{P_0} equals R instead of 1. Those data are then transmitted to the clients by any compliant mean. After the computation of a phase 1, z_Q is retrieved and a $\gcd()$ is computed. The factor R must also be lifted from z_Q by a division.

Those computations are achieved by a C program. It relies on the easy-to-use and highly efficient GNU Multiple Precision Arithmetic Library[3] (GMP).

If the cost of the server's task appears to be non negligible, a small arithmetic processor can still be included in the hardware to take care of those computations.

5.3 Hardware Client

Concerning phase 1, the hardware client computes a scalar multiplication for each curve. According to Algorithm 3 and 4, each step implies the computation of both $2P$ and $P + Q$. Previous points P and Q or Q and P are then overwritten by those new points, according to the current bit of the constant scalar k (pre-stored in memory).

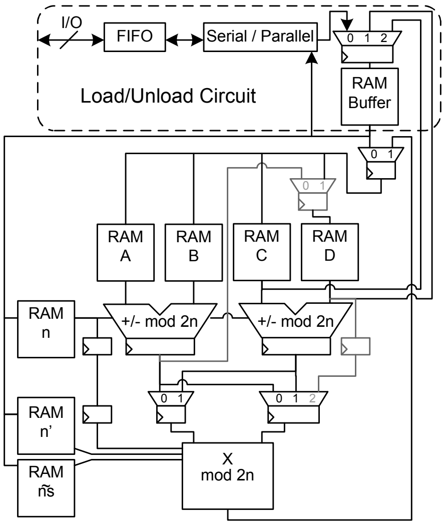

Figure 4. Architecture of ECM, Phase 1

The circuit implementing phase 1 is presented in Figure 4. Besides the two add/sub circuits and the modular multiplier, memory and a simple load/unload architecture to communicate with the server are needed. For each of the 4 RAM banks A-B-C-D, each curve owns 2 memory locations reachable with the same addresses. The memory stores the coordinates of points P and Q, a_{24}, x_{P-Q} and

[3] Available at `http://www.swox.com/gmp/`

intermediate results. Those data stand in the 8 possible locations in such a way that each input of both add/sub circuits is directly fed by a RAM. This means no multiplexer is required. There are more intermediate results than the 2 remaining locations but some can be overwritten. A particular case is the computation of M_{10} (see below) where $(M_2 - M_1)$ cannot be re-computed later. This subtraction is stored in the D RAM bank and bypasses the add/sub to not add an extra 2^{-2} factor (shaded elements of Figure 4).

Each of the A, B, C, D, n, $\widetilde{ns}$ RAM bank is composed of $\lceil d/2 \rceil$ (4) parallel Virtex4 block RAMs (bRAMs) of 34-bit width. Two extra 17-bit registers are used to retime the read and write of the two digits held in each 34-bit bRAM. The global temporal shift of the digits is taken into account for the read/write operations of each RAM bank. The n' RAM is different since it stores only one digit.

The purpose of the two multiplexers after the two add/sub circuits is to allow the modular squaring operation.

Due to the latency of the arithmetic core, data-dependency problems occur during computation of Algorithm 3. Three independent steps can be extracted:

$$\begin{array}{cl}
\text{I} & M_1 \leftarrow (x_P - z_P)^2,\ M_3 \leftarrow (x_P - z_P) \cdot (x_Q + z_Q), \\
 & M_2 \leftarrow (x_P + z_P)^2,\ M_4 \leftarrow (x_P + z_P) \cdot (x_Q - z_Q) \\
\text{II} & M_5 \leftarrow M_1 \cdot M_2,\ M_7 \leftarrow (M_3 + M_4)^2, \\
 & M_6 \leftarrow (M_2 - M_1) \cdot a_{24},\ M_8 \leftarrow (M_3 - M_4)^2 \\
\text{III} & M_9 \leftarrow x_{P-Q} \cdot M_8,\ M_{10} \leftarrow (M_2 - M_1) \cdot (M_1 + M_6)
\end{array}$$

To cope with this dependency problem, computation on another set of data (other curves) is started until completion of the computation of the first set. With operands up to 135 bits, 32 curves running simultaneously are enough to ensure that the pipeline is always filled.

In order to feed the FPGA and transmit the results back to the server, a load/unload circuit is used (see Figure 4). A FIFO (made of 2 bRAMs) buffers the communications while a simple shift register deserializes/serializes the inputs/outputs. In order to reduce the overheads, a buffer (made as a RAM bank) stores the results and the new data during the computation of a phase 1. This renders negligible the delay between two phase 1 computations. For each curve, the new data is a set of 8 values of $17d$ bits (136) and the result consists in the point Q (x_Q, z_Q, located in the C and D RAMs).

5.4 Phase 2

This work focuses on phase 1. As a circuit for phase 2 can be implemented with a similar structure, provided results are sufficient to show the effectiveness of the proposed method. The main difference is the need of extra RAM banks for precomputed values x_{jQ} and extra intermediate results. As the computations are less specific, more multiplexors are also required to connect the different RAM banks to the add/sub circuits.

The high level control of phase 2 lies in the iterative lookup of table MJ (Algorithm 2). As the number of tries before finding the condition $MJ[m, j] = 1$ is smaller than the latency of the arithmetic core, no pipeline stalls occur.

6 Implementation Results

Implementation results were achieved for the smallest Xilinx Virtex4 SX FPGA with the lowest speedgrade. ISE 8.1 was used for synthesis and place & route while test/debug was performed with Modelsim SE 6.1. For the real tests, the setup was a Pentium4 3.2 Ghz desktop computer running WindowsXP for the server and an Avnet *Virtex-4 SX35 Evaluation Kit* for the client. For testing purposes, a slow RS232 communication between the PC and the FPGA was used. A more appropriate fast connection like USB2 should be used.

6.1 Software

In order to have a consistent Server-Client model, the software server must be able to feed many hardware clients. This puts the cost of the hardware in foreground, as expected. While in [4] the server overhead is very small, it appears that our hardware is so fast that precomputations and communication are not negligible at all.

For the communication, a burst of 32 data sets has to be sent. For the 125-bit inputs of SHARK, each data set requires $7 \times 125 + 17$ bits for the a_{24}, P_0, $2P_0$, $\widetilde{ns}$, n and n' values. Assuming a 32-bit data bus and a rate of 16,000 phases 1 per second, a bandwidth of 15Mb/s is required. If all the precomputation is performed in hardware, this requirement can fall to 2Mb/s. When the computation is completed, only one value has to be sent back. This corresponds to a bandwidth of 2Mb/s.

The slowest precomputation operations are the $\gcd()$ and the 4 modular inversions required for the computation of a_{24}, n', the normalization of x_P and lifting of R in z_Q. Other significant operations are the 25 multiplications. Multiplication is roughly 10 times faster than the inversion on the selected software platform. With our setup, the PC has the computational power to only deal with 4 FPGAs. A low-throughput arithmetic processor with support for modular division and multiplication should therefore be included in the FPGA. It should not pose any particular problem as many slices are available (roughly 4000 from Table 2). Phase 2 will definitively need hardware support for precomputation and in particular for the table T (set of jQ).

6.2 Hardware Arithmetic Operators

The arithmetic operators were implemented for numbers up to 135 bits, a little more than the SHARK requirements. As all the circuits are pipelined, moving to other lengths does not raise any particular issue. With this length, the operands are represented by 8 digits and the Montgomery constant R equals $2^{1+17\cdot 8} \mod n$. This means that operands must be premultiplied by $2^{4+1+17\cdot 8} \mod n$.

The implementation results are given in Table 1. For the modular multiplier, the maximum frequency reaches 220 MHz despite the 100% utilization of the DSP multipliers. The area requirements are satisfying since almost 66% of the slices are available for the rest of the ECM processor. The arithmetic operators produce an output per clock cycle.

135-bit Designs	Frequ. [Mhz]	Area [Slices]	Through. [Gbit/s]	DSP48
Mult.	220	3405 (33%)	30	128 (100%)
Add/Sub	245	446 (4%)	33	0

Table 1. Implementation results on a XC4VSX25-10

6.3 ECM Architecture

Phase 1 was implemented for 135-bit numbers which is different from the 198-bit numbers used by Gaj et al. [4] and Pelzl, Šimka et al. [14]. Nevertheless, a comparison of implementation results can be done on basis of extrapolations since the size of their design varies linearly with the size of the numbers. For 135-bit numbers, the size of their design is approximately decreased by a factor $\frac{198}{135}$ and more ECM units can therefore fit on an FPGA. The running time is also shortened because of the serial by parallel architecture. E.g. a modular multiplication is then performed in $135 + 16$ clock cycles in place of $198 + 16$ (for [4]).

The impact of this bit-size scale down on the operating frequency is more difficult to evaluate. However, as the design of Gaj et al. is scalable and uses carry save adders, it is assumed that the operating frequency will not vary. Compared to the design by Pelzl, Šimka et al., results of Gaj et al. exhibit an improvement factor of 3.4 in terms of throughput/hardware cost ratio for phase 1. Only this factor will be used for further comparison with the design of Šimka et al.

In order to achieve the best throughput/hardware cost ratio, the cheapest FPGA able to hold the modular multiplier was chosen: the Virtex4SX25-10. For other bit-sizes, 2007/2008 Xilinx's prices for FPGAs of the same family are $183 for the SX35 and $454 for the SX55 ($2500$ devices). Using the *EasyPath*[4] solution those prices can be lowered even further: $73 for the SX35 ($10^4$ devices) and $230 ($5\ 10^3$ devices) for the SX55 with both a NRE cost of $73000. Prices for low-cost Spartan3 FPGAs range from $20 to $130 ($10^4$ devices). The biggest of this family was chosen by Gaj et al.

Phase 1, 135-bit	Gaj et al. [4]	Our design
FPGA	XC3S5000-5	XC4VSX25-10
Slices / ECM unit	2300 (7%)	6006 (58%) (+ 128 DSP48)
bRAMS / ECM unit	2 (2%)	31 (24%)
Max frequency	100 MHz	220 MHz
# T_{clk} / Mod. Mult.	151	1
# T_{clk} / phase1	$1.22\ 10^6$	13,750
# phases1/sec. / ECM unit	82	16,000
# ECM units / FPGA	14	1
# of phases1/sec. / FPGA	1148	16,000
FPGA price (quantity)	$130 ($10^4$)	$116 (2500)
# of phases1/sec. / $100	883	13,793
Improvement factor		**15.6**

Table 2. Comparison of implementation results

The comparison of performances for phase 1 is given in Table 2. The size of the whole ECM processor is dominated by the modular multiplier (cf. Table 1). In terms of throughput/hardware cost ratio, our design outperforms the architecture of Gaj et al. by a factor $\frac{13793}{883} = \mathbf{15.6}$ and the design of Pelzl, Šimka et al. by a factor $15.6 \cdot 3.4 = \mathbf{53}$. This big improvement factor really suggests using high-performances FPGAs and embedded multipliers instead of general purpose logic.

6.4 Cost estimate for a sieving device

A significant improvement in the design of the ECM processor has a great impact when considering the implementation of the entire NFS sieving step. Based on results of Section 6.3 a quick cost assessment can be provided. Even if phase 2 of the ECM algorithm was not explicitly implemented, it is claimed that a similar structure as for phase 1 can be used. The running time of phase 2 should be about $13k$ clock cycles since it is roughly the number of modular multiplications. Assuming a frequency of 220 MHz can be reached as for phase 1, phase 2 should be performed ca. $\frac{2.2\ 10^8}{13\ 10^3} \approxeq 16,900$ times per second per FPGA (i.e. $116).

From [3], 10^{14} up to 125-bit numbers must be factored in the sieving step of SHARK. If this task is distributed over a year, approximately $5.5\ 10^6$ of these numbers must be factored per second. According to [14], 20 curves per number must be used to have a probability of success $> 80\%$. Unfortunately the success rate of the two phases is not independently provided. The worst case – meaning phase 2 is always performed – is assumed in order to obtain a higher

[4] Xilinx's EasyPath means the FPGA correct behavior is only guaranteed for a given configuration.

bound for the cost estimate. In a second, $5.5\ 10^6$ factorizations require $20 \cdot 5.5\ 10^6 = 1.1\ 10^8$ phases 1 and phases 2 which are respectively achieved by $\frac{1.1\ 10^8}{16\ 10^3} \approxeq 6900$ and $\frac{1.1\ 10^8}{16.9\ 10^3} \approxeq 6500$ FPGAs.

The overall purchase cost of FPGAs for the ECM factorizations of the sieving step is approximatively $(6900 + 6500) \cdot 116 \approxeq \$1.6$ M. Roughly speaking, this price could even be halved using SX55 FPGAs (with 4 ECM units each) and the EasyPath solution. Using the design of Gaj et al., the same computations (135-bit inputs) – with about 1148 phases 1 and 1075 phases 2 per second – gives an overall cost of roughly $\$25.8$ M.

7 Conclusion and Further Work

This work presented a novel hardware architecture for implementing ECM on FPGA. The motivation was to assess the cost of an ECM processor as a support for the NFS algorithm. The aim was also to show how to better exploit resources of reconfigurable hardware platforms.

For numbers in the context of SHARK device (135-bit) the throughput/cost ratio of the phase 1 was improved by a factor **15** with respect to the best published results.

Further work on this topic should include the implementation of phase 2 and a small processor for the precomputation. Digit-serial architecture could also be built, taking this work as a benchmark for performances. Finally, figures like power consumption of FPGAs and cooling, cost of a complete board, expenses for housing and servers could be included to achieve a complete cost assessment.

For the implementation of ECM, the big improvement achieved in this paper really suggests using embedded multipliers and high-performances FPGAs (with a high multiplier density) instead of general purpose logic and low-cost FPGAs.

References

[1] I.F. Blake, G. Seroussi, N.P Smart, *Elliptic Curves in Cryptography*, London Mathematical Society, LNS 265, Cambridge University Press, 1999.

[2] R. Brent, Recent Progress and Prospects for Integer Factorisation Algorithms, *COCOON'00*, LNCS 1858, Springer-Verlag, pp. 3-22, 2000.

[3] J. Franke, T. Kleinjung, C. Paar, J. Pelzl, C. Priplata, C. Stahlke, SHARK: A Realizable Special Hardware Sieving Device for Factoring 1024-bit Integers, *CHES'05*, LNCS 3659, Springer, pp. 119-130, 2005.

[4] K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, R. Bachimanchi, Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware, *CHES'06*, LNCS 4249, Springer, pp. 119-133, 2006.

[5] W. Geiselmann, F. Januszewski, H. K opfer, J. Pelzl, R. Steinwandt, *A Simpler Sieving Device: Combining ECM and TWIRL*, ICISC'06, LNCS, Springer, 2006.

[6] W. Geiselmann, A. Shamir, R. Steinwandt, E. Tromer, Scalable Hardware for Sparse Systems of Linear Equations, with Applications to Integer Factorization, *CHES'05*, LNCS 3659, Springer, pp. 131-146, 2005.

[7] L. Hars, Long Modular Multiplication for Cryptographic Applications, *CHES'04*, LNCS 3156, pp. 44-61, 2004.

[8] T. Kleinjung, Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024 bit integers, *SHARCS'06*, Workshop on Special Purpose Hardware for Attacking Cryptographic Systems, pp. 159-168, 2006.

[9] H. Lenstra, Factoring integers with elliptic curves, *Annals of Mathematics*, Vol. 126, pp. 649 - 673, 1987.

[10] A.K. Lenstra, H.W. Lenstra, The Development of the Number Field Sieve, *Lecture Note in Math.*, Vol. 1554, Springer-Verlag, 1993.

[11] P. Montgomery, Modular Multiplication without Trial Division, *Mathematics of Computation*, No. 44(170), pp. 519-521, 1985.

[12] P. Montgomery, Speeding the Pollard and elliptic curve methods of factorization, *Mathematics of Computation*, 48(177), pp. 243 - 264, 1987.

[13] H. Orup, Simplifying Quotient Determination in High-Radix Modular Multiplication, *ARITH-12*, IEEE, pp. 193-199, 1995.

[14] J. Pelzl, M. Šimka, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovsky, V. Fisher, C. Paar, Area-time efficient hardware architecture for factoring integers with the elliptic curve method, *IEE Proceedings on Information Security*, Vol. 152, No. 1, pp. 67-78, 2005.

[15] C. Pomerance, A tale of Two Sieves, *Notices of the AMS*, pp. 1473-1485, 1996.

[16] R.L. Rivest, A. Shamir, L.M. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *Communications of the ACM*, Vol. 21, No. 2, pp. 120-126, 1978.

[17] *SHARCS'05*, Workshop on Special Purpose Hardware for Attacking Cryptographic Systems, Paris, 2005.

[18] S.H. Tang, K.S. Tsui, P.H.W. Leong, Modular exponentiation using parallel multipliers, *FPT'03*, IEEE, pp. 52–59, 2004.

[19] C.D. Walter, Precise Bounds for Montgomery Modular Multiplication and Some Potentially Insecure RSA Moduli, *CT-RSA'02*, LNCS 2271, pp. 30-39, Springer, 2002.

[20] P. Zimmermann, 20 years of ECM, *ANTS VII*, LNCS 4076, Springer, pp. 525-542, 2006.